Visualization of Multithreaded Behavior to Facilitate Maintenance of Complex Software Systems

Jonas Trümper, Johannes Bohnet, Stefan Voigt and Jürgen Döllner
{jonas.truemper}, {johannes.bohnet}, {stefan.voigt}, {juergen.doellner}@hpi.uni-potsdam.de
Hasso-Plattner-Institute – University of Potsdam, Germany

***Abstract*—Maintenance accounts for the major part of a software system's total costs. Therein, program comprehension is an important, but complex activity: Typically, up-to-date documentation is not available, so the main reliable source of information on the implementation represent the artifacts of the system's implementation. Understanding software systems is difficult, in particular, if multithreading concepts are involved because state-of-the art development tools provide only limited support for maintenance activities. In addition, concurrency is often not directly reflected by the source code, i.e., there is only a non-obvious correlation between control structures in the source code and a system's runtime behavior. We present a program comprehension technique that helps to analyze and understand runtime behavior of multithreaded software systems and, thereby, facilitates software maintenance tasks. Our approach contains the following concepts: First, light-weight dynamic analysis records executed method calls at runtime. Second, visualization of multithreading trace data allows developers to explore the system behavior post-mortem. The technique forms part of a scalable tool suite for understanding the behavior of complex software systems. We also show how to apply the technique on industrial software systems to solve common maintenance problems.**

I. Introduction

Software maintenance represents an essential, time- and cost-intensive task for the management of complex software systems [14]. It accounts for the major part of a software system's total costs: "Year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little" [18]. Maintenance is typically difficult because, among many reasons, these systems (1) are large, e.g., comprise more than 500k lines of code (LOC), (2) are weakly understood, (3) are hardly documented, and (4) often exhibit substantial differences between the existing documentation and the as-is system design due to the system's long evolution period [5] [14]. Hence, maintenance significantly requires program understanding, i.e., developers build a mental map of the system's functionality and concepts based on its implementation artifacts [10]. Building such a mental map typically involves studying existing source code and design documents as well as stepping through the execution using a debugger to uncover and understand the original design concepts and the system's behavior. As a result, 50-60% of the time used for software maintenance tasks is actually spent for program comprehension [1] [20].

Understanding source code of multithreaded systems is often complicated because a static view on control structures, e.g., by analyzing code, does not explicitly reveal runtime concurrency. In addition, debuggers and profilers, while being established tools for debugging and testing software having predominantly single-threaded runtime behavior, typically provide limited support for multithreaded software: There are issues related exclusively to concurrent systems that are insufficiently supported by today's tools [4], e.g., deadlocks, load imbalance, data sharing patterns, race conditions, or contention. Existing tools mostly ignore timing and scheduling, which have to be considered in concurrent systems.

To circumvent today's lack of appropriate tools, developers have established workarounds such as manual source code augmentation. For example, console debug output can be triggered, but is a resource-intensive operation, thus likely to cause noticeable performance decrease or to change the system's timing essentially. Since source code augmentation also is a time consuming and tedious task, developers frequently apply error-prone guesswork instead. As a result, modifications "made by people who do not understand the original design concept almost always cause the structure of the program to degrade" [14].

In this paper we present a program comprehension technique that helps to understand the runtime behavior of complex, multithreaded software. It selectively instruments binaries of the analyzed software system to record execution traces at runtime. These traces are postprocessed and imported into the analysis tool. The proposed visualization technique generates stack based, synchronized views on sequences of method[1] calls for multiple threads. The technique enables developers to interactively explore the traces. The tool has been implemented as a plugin of the *Software Diagnostics Developer Edition*—a commercial tool for tracing and visualizing runtime behavior of software systems and has been tested with software systems consisting of over 4 million LOC. We demonstrate its usage and benefits in maintenance by two case studies.

II. Related Work

Existing techniques for dynamic analysis of concurrent behavior primarily focus on performance optimization or coarse-grained behavior visualization and have shortcomings with respect to program comprehension [4]. Examples of such shortcomings include insufficient support for interactive exploration of execution traces at varying levels of detail and scalability issues.

[1]From now on, the term method is being used interchangeably with the terms function, procedure, routine etc.

Early work on analysis of multithreaded runtime behavior was done by Malony and Reed (1989) [11]. Their approach focuses on monitoring and collecting statistics like communication bandwidth. Heath (1993) [7], Nutt et al. (1995) [13] and Nagel et al. (1996) [12] propose tools that aim at performance optimization for parallel message passing systems. In contrast to our technique, these approaches target analysis of process interactions, the processes often running on separate machines.

Stasko (1995) [16] introduces PARADE, which partially targets program comprehension. However, the tool visualizes multiple threads only separately and thus developers are unable to tell which activities in separate threads actually happen in a concurrent manner. Kergommeaux and Stein (2000) [8] as well as Bedy et al. (2000) [2] propose a visualization that depicts 2-dimensional graphs with life lines for each thread showing their state (running, suspended, etc.). Contrary to our approach, this graph visualization does not show the threads' execution contexts or call stacks. Reiss (2007) [15] presents live visualization systems that depict an analyzed system's behavior while the system is executed. These approaches differ from our technique in that they provide only coarse-grained information on the system's execution. Wheeler and Thain (2009) [19] propose ThreadScope, a tool for identifying structural and synchronization problems. The generated 2-dimensional graphs do not scale well for large execution traces, and there is, in contrast to our approach, potential for interactive exploration, which would allow for enhanced scalability of their tool.

Existing work for execution trace summarization and abstraction techniques that enable developers to assess a trace's essence, by contrast to our approach, focus on sequential execution [3] [6] [9].

With our approach, we aim to improve on existing techniques and tools by enabling developers to analyze multiple threads' execution contexts and call stacks in parallel in a synchronized manner. Furthermore, our approach improves on existing work in terms of scalability.

III. Analysis Process

The analysis process of our technique consists of two phases:

1) Tracing: The analyzed software system is instrumented. During execution of the system, a *trace* is recorded.
2) Visualization and exploration: Recorded trace data is imported into the analysis tool. First, developers select the threads to be visualized in a *textual thread overview*. Then, the interactive *sequence visualization* is rendered and can be used for exploration.

IV. Tracing

A. Properties of Trace Data

For each thread i, the recorded *trace* essentially forms an ordered group E_i of *timestamped events* e_j (with j being the timestamp). That is, $\forall\ e_j, e_k \in E_i$, e_j happens before e_k if $j < k$. Events $e \in E_i$ are typed: They are either of type *method entry* or *method exit*. Furthermore, each event comprises (1) the address of the respective method being called or returning and (2) the address of the calling method.

B. Minimizing Runtime Overhead

Generally, instrumentation causes a runtime overhead which may cause unintended or unexpected behavior of the instrumented software system if timing is critical. In multithreaded software systems, the instrumentation may alter the usual system behavior. For instance, instrumentation may cause (or prevent) deadlocks or race conditions. To minimize such interferences, we (1) minimize the runtime overhead of the instrumentation routines and (2) instrument only those parts of the system implementation that are relevant for the maintenance task at hand.

Developers can specify at runtime which parts of the implementation are actually instrumented. Hence, they initially run the system with full instrumentation and record a trace. By roughly exploring the resulting trace, they identify those parts of the implementation that are relevant for the given task and disable all irrelevant methods for further tracing. Next, they exercise the system again with only selective instrumentation.

In our case studies, the technique has not caused interferences. However, the technique has natural limitations if highly time critical systems, e.g., embedded real-time systems in the automotive domain, are to be analyzed.

V. Visualization

Traces typically contain hundreds or thousands of method calls. To explore such traces, developers need effective visualization that presents trace data in an interactive way. Additional orientation aids, e.g., by means of overviews, in the mass of data are considered helpful [3] [6] [9].

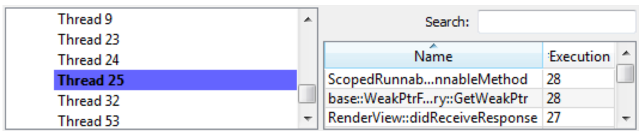
A. Concept

Multithreaded software systems may spawn numerous threads at runtime. To ease exploration of their behavior, a means of choosing a representative subset of all threads is required. Our analysis tool provides a *textual thread overview* of the activities for selected threads, which shows all methods invoked in the context of the respective thread including their invocation count (Fig. 1). This way, developers find and choose representative threads to analyze.

After choosing the threads, trace data is shown in a *visual thread overview* and in a *sequence view* for each thread separately. These views are synchronized, which permits the analysis of each thread's activity separately and in parallel.

Each sequence view depicts a thread's activity using a 2-dimensional graph representation. The execution sequences are visualized using a stack based approach. That is, time is mapped on the x-axis and call stack along the y-axis (Fig. 2). Call relations between functions are implicitly expressed. That is, if function *foo* calls *bar*, then *bar* is drawn below *foo*. Since the complete sequence graph typically exceeds available screen space, panning and zooming functions are provided so that developers can adjust the shown time span to their needs.

To scale with large execution traces, the visualization technique makes use of out-of-core concepts: Only a sub time span



Name	Execution
ScopedRunnab...nnableMethod	28
base::WeakPtrF...ry::GetWeakPtr	28
RenderView::didReceiveResponse	27

Fig. 1. Textual thread overview: Depicting method executions in the context of a selected thread.

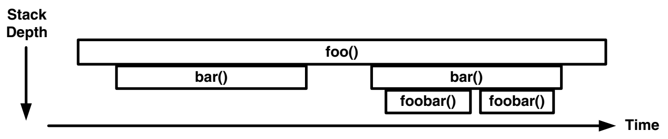


Fig. 2. Illustration of the interactive sequence visualization for a single thread.

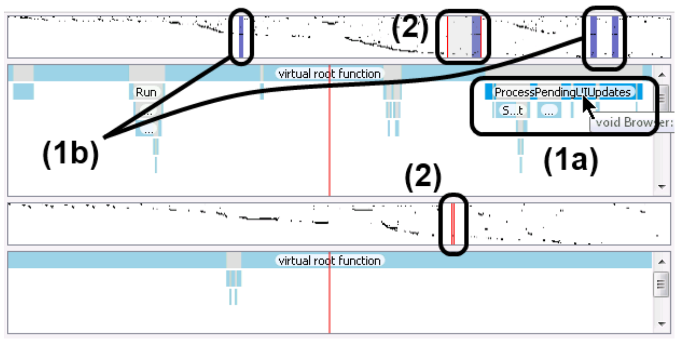


Fig. 3. Sequence visualization showing 2 threads. Activities of methods hovered in the sequence visualization (1a) are highlighted in each visual thread overview (1b). The sub time span (in the sequence visualization) is marked according to the respective visual thread overview (2).

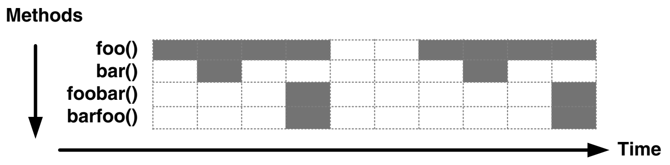


Fig. 4. Illustration of a visual thread overview: Repeated executions of the same (or similar) functionality causes repeating visual patterns.

of the complete recorded time span is kept in main memory and is depicted as detailed sequence. Additional *visual thread overviews* are generated once, facilitating quick orientation within the trace data. The current detailed sequence sub range is marked within the respective visual thread overview (Fig. 3).

Ware [17] states that “the human visual system is a pattern seeker of enormous power”. To exploit this power, the visual thread overview converts execution patterns to visual patterns: A 2-dimensional grid is computed where methods are mapped along the grid’s y-axis and time is mapped along the x-axis. When a method is executed at a specific point in time, the respective grid cell is colored black, white otherwise. This way, repeated execution of the same (or similar) functionality causes repeating visual patterns in an overview (Fig. 4).

B. Synchronizing Multiple Sequence Views

A key challenge when building a trace visualization tool for multiple threads is the way multiple views—each depicting single thread activity—can be synchronized such that collaboration between threads gets visible. Without effective synchronization, developers need to compare event timestamps manually to assess whether a specific method execution in the context of a thread is actually concurrent to another method execution in the context of a second thread.

Furthermore, execution durations of methods typically vary significantly: Where execution of method *foo* may take 10,000 milliseconds, the execution of method *bar* may take only a few milliseconds. Even for the visualization of a single thread’s activity, this poses a challenge: Using these raw execution durations, long lasting method executions would span multiple screens whereas a very short execution might span only a single pixel (Fig. 5a). Consequently, a time distortion is required for exploration and can be implemented by a *scaling* transformation.

We apply a logarithmic-based scaling for an execution’s time interval Δt. The scaling shrinks short time spans only slightly, but long time spans massively:

$$logscale(\Delta t) = \begin{cases} 1 & \text{if } \log(\Delta t) < 1 \\ log(\Delta t) & \text{if } 1 < \log(\Delta t) \leq \text{const} \\ const & \text{if } \log(\Delta t) > \text{const} \end{cases}$$

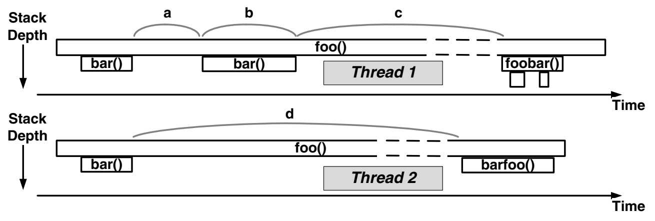
Applying this non-linear scaling function *logscale* on each threads’ events would break visual comparability between concurrent method executions. Method executions or idle times of separate threads (time spans a, b, c and d in Fig. 5a) that happen before *foobar* and/or *barfoo* would be scaled differently (Fig. 5b), resulting in warped method entry and exit timestamps of *foobar* and *barfoo*.

To solve this issue and to preserve concurrency of method executions, we apply on-demand scaling for the selected subset of threads and for the currently depicted sub time span. That is, we analyze all events in the selected time span for all selected threads and identify *concurrency constraints*, i.e., pairs of concurrent method executions, that need to be satisfied (Fig. 5c). Thus, we yield an event order in which we can apply the scaling function while preserving existing concurrency of method executions (Fig. 5d). Our approach then both scales time and synchronizes multiple views by focusing on a specific point in time and by updating all views such that the views are centered on this point in time.

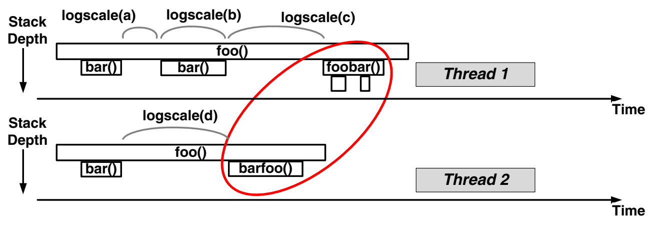
VI. Case Studies

We discuss two case studies that we have performed on industrially developed and maintained software systems: (1) *Chromium*[2], the open-source code base of Google’s web browser Chrome and (2) *BRec* (Building Reconstruction), a tool for reconstructing 3D building models from laser scan data, developed and maintained by virtualcitySYSTEMS GmbH, one of our industrial partners.

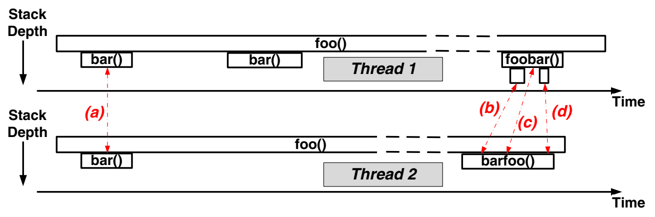
[2]http://www.chromium.org, last accessed 10.04.2010



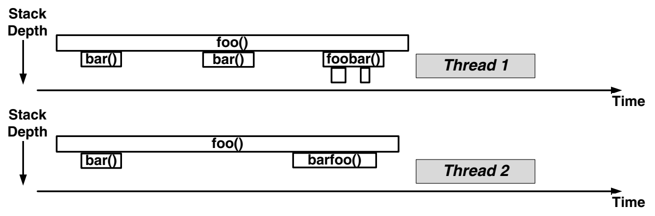
(a) Raw time stamps used. Note the large variations in execution durations of, e.g., *foo()* and *bar()*.



(b) Separate logarithmic scaling per thread. Note that the execution of *foobar()* in thread 1 is no longer concurrent to the execution of *barfoo()* in thread 2.



(c) Excerpt of the concurrency constraint set (*a*, *b*, *c* and *d*) derived from raw event timestamps.



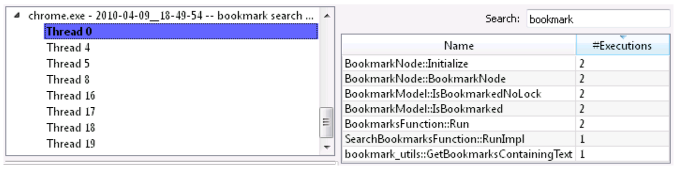
(d) Logarithmic scaling for all selected threads with concurrency constraints satisfied: concurrency of *foobar()* and *barfoo()* is preserved.

Fig. 5. Concurrency issue with logarithmic scaling and its solution.

A. *Chromium - Opening a Website via Bookmark*

Chromium consists of approximately 4 million LOC[3]. The source code repository lists 388 authors as contributing to the code base. The implementation concepts of Chromium strongly rely on multithreading, e.g., tasks are 'posted' at runtime and successively solved by dedicated threads. Hence, developers that need to understand the internal workings of the 4 MLOC system Chromium likely face understanding problems that are also faced during maintenance of complex closed-source multithreaded systems.

In our case study, we needed to understand how Chromium's threads collaborate when users issue a bookmark search and open the respective website. For this task, we identified relevant parts of the implementation (Section IV-B) and instrumented the system partially for a second run. We exercised the bookmark scenario and imported the resulting trace data (approx. 200,000 events) into our analysis tool.



chrome.exe - 2010-04-09__18-49-54 -- bookmark search ...
Thread 0
Thread 4
Thread 5
Thread 8
Thread 16
Thread 17
Thread 18
Thread 19

Search: bookmark

Name	#Executions
BookmarkNode::Initialize	2
BookmarkNode::BookmarkNode	2
BookmarkModel::IsBookmarkedNoLock	2
BookmarkModel::IsBookmarked	2
BookmarksFunction::Run	2
SearchBookmarksFunction::RunImpl	1
bookmark_utils::GetBookmarksContainingText	1

Fig. 6. Chromium: Identifying threads that are relevant for the comprehension task in the textual thread overview.

With the textual thread overview, 2 out of 8 threads were identified as relevant for our comprehension task (Fig. 6). Next, the sequence visualizations and the visual thread overviews showed each thread's overall activity. After initially seeking through the trace data, we recognized relevant parts therein by their patterns in the visual thread overview and navigated using the pan and zoom tools. We identified 3 key points in the trace data that are of high relevance for our comprehension task:

1) After the bookmark search term is entered in Chromium, a message is sent as an IPC and received by the thread responsible for bookmark searches ((1) in Fig. 7): *RenderViewHost::onExtensionRequest*. The thread performs the search (2) (*bookmark_utils::GetBookmarksContainingText*) and sends its response (3) which is then received by another thread (4).
2) Before the bookmarked page is actually loaded in a new tab, Chromium updates its history asynchronously via an IPC (Fig. 8).
3) The new tab is created: A new *RenderView* (that contains the rendered page) is created asynchronously (Fig. 9). Internally, Chromium uses the Webkit[4] browser engine for rendering. In this context, Chromium prepares everything needed to render a new page and then hands over rendering to Webkit.

Chromium makes heavy use of asynchronous task scheduling. The proposed technique facilitates (1) program comprehension as task processing responsibility becomes visible and (2) debugging in case of erroneous task scheduling as it permits to reconstruct when a specific task is processed.

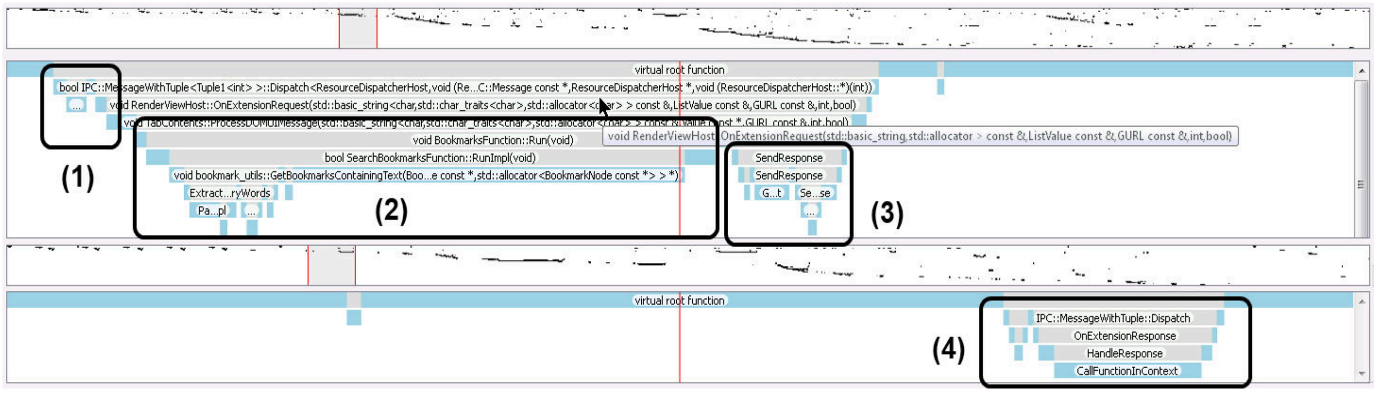
B. *BRec - Simplifying the Rendering Process*

BRec reconstructs 3D building models from given laser scan data. A key characteristic of BRec's rendering engine is that the triangulation of building models is parallelized. BRec comprises approx. 100k LOC written in C/C++.

The task to be solved was to understand and simplify BRec's rendering functionality. The developer started with recording a trace of the rendering functionality (approx. 100,000 events). With the textual thread overview, he identified one coordinating thread and a number of worker threads that all execute the

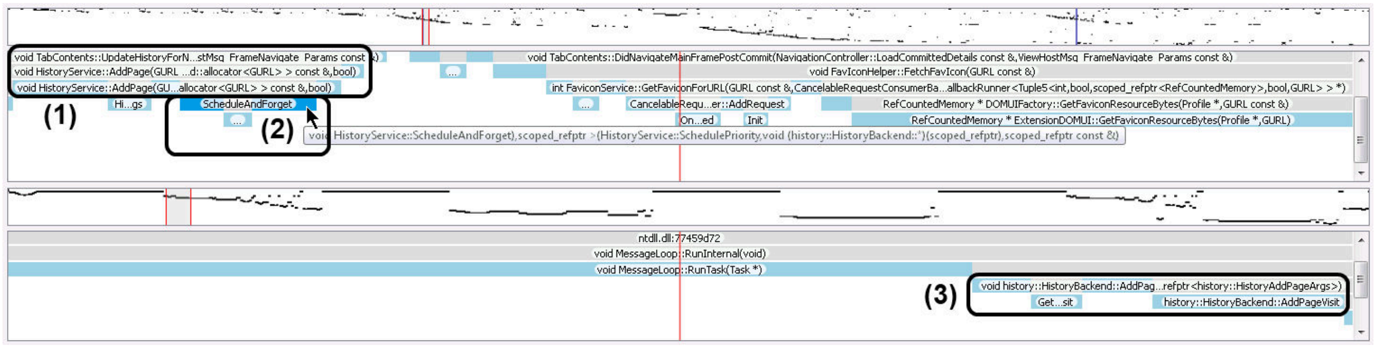
[3]http://www.ohloh.net/p/chrome/analyses/latest, last accessed 10.04.2010

[4]http://webkit.org, last accessed 10.04.2010



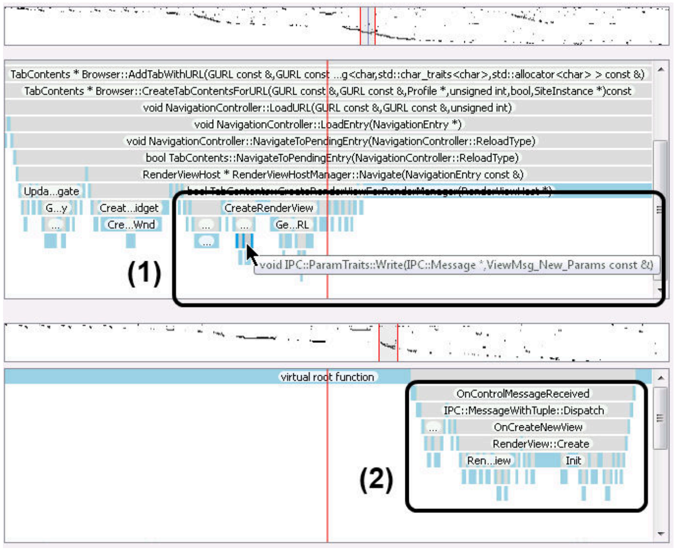
virtual root function
bool IPC::MessageWithTuple<Tuple1<int> >::Dispatch<ResourceDispatcherHost,void (Re...C::Message const *,ResourceDispatcherHost *,void (ResourceDispatcherHost::*)(int))
void RenderViewHost::OnExtensionRequest(std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,ListValue const &,GURL const &,int,bool)
void TabContents::ProcessDOMUIMessage(std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,Value const *,GURL const &,int,bool)
void BookmarksFunction::Run(void)
void RenderViewHost::OnExtensionRequest(std::basic_string,std::allocator > const &,ListValue const &,GURL const &,int,bool)
bool SearchBookmarksFunction::RunImpl(void)
void bookmark_utils::GetBookmarksContainingText(Boo...e const *,std::allocator<BookmarkNode const *> > *)
Extract...ryWords
Pa...pl
SendResponse
SendResponse
G...t
Se...se
(1)
(2)
(3)
virtual root function
IPC::MessageWithTuple::Dispatch
OnExtensionResponse
HandleResponse
CallFunctionInContext
(4)

Fig. 7. Sequence visualization of Chromium bookmark search: (1) receive request as IPC, (2) process request: search in bookmarks, (3) send response as IPC, (4) receive response as IPC.



void TabContents::UpdateHistoryForN...stMsg_FrameNavigate_Params const &)
void HistoryService::AddPage(GURL ...d::allocator<GURL> > const &,bool)
void HistoryService::AddPage(GU...allocator<GURL> > const &,bool)
Hi...gs
ScheduleAndForget
void TabContents::DidNavigateMainFramePostCommit(NavigationController::LoadCommittedDetails const &,ViewHostMsg_FrameNavigate_Params const &)
void FavIconHelper::FetchFavIcon(GURL const &)
int FaviconService::GetFaviconForURL(GURL const &,CancelableRequestConsumerBa...allbackRunner<Tuple5<int,bool,scoped_refptr<RefCountedMemory>,bool,GURL> > *)
CancelableRequ...er::AddRequest
RefCountedMemory * DOMUIFactory::GetFaviconResourceBytes(Profile *,GURL const &)
On...ed
Init
RefCountedMemory * ExtensionDOMUI::GetFaviconResourceBytes(Profile *,GURL)
void HistoryService::ScheduleAndForget),scoped_refptr >(HistoryService::SchedulePriority,void (history::HistoryBackend::*)(scoped_refptr),scoped_refptr const &)
(1)
(2)
ntdll.dll:77459d72
void MessageLoop::RunInternal(void)
void MessageLoop::RunTask(Task *)
void history::HistoryBackend::AddPag...refptr<history::HistoryAddPageArgs>)
Get...sit
history::HistoryBackend::AddPageVisit
(3)

Fig. 8. Sequence visualization of Chromium updating its history: (1) trigger update functionality, (2) insert history update into a task queue, (3) history update task is processed in another thread.



TabContents * Browser::AddTabWithURL(GURL const &,GURL const ...g<char,std::char_traits<char>,std::allocator<char> > const &)
TabContents * Browser::CreateTabContentsForURL(GURL const &,GURL const &,Profile *,unsigned int,bool,SiteInstance *)const
void NavigationController::LoadURL(GURL const &,GURL const &,unsigned int)
void NavigationController::LoadEntry(NavigationEntry *)
void NavigationController::NavigateToPendingEntry(NavigationController::ReloadType)
bool TabContents::NavigateToPendingEntry(NavigationController::ReloadType)
RenderViewHost * RenderViewHostManager::Navigate(NavigationEntry const &)
Upda...gate
bool TabContents::CreateRenderViewForRenderManager(RenderViewHost *)
G...y
Creat...idget
Cre...Wnd
CreateRenderView
Ge...RL
void IPC::ParamTraits::Write(IPC::Message *,ViewMsg_New_Params const &)
(1)
virtual root function
OnControlMessageReceived
IPC::MessageWithTuple::Dispatch
OnCreateNewView
RenderView::Create
Ren...iew
Init
(2)

Fig. 9. Sequence visualization of Chromium creating a new *RenderView* for the new tab we opened from the bookmark search.

same utility functionality: Delaunay triangulation. The coordinating thread is identified as it executes the main window's event loop and exhibits significantly higher execution counts than the worker threads.

The developer selected the coordinating thread and two worker threads for visualization. He inspected the visualized sequences and identified the following behavior within the coordinating thread: Delaunay triangulation is triggered from constructors of *SolidRenderer* ((1a), (1b) in Fig. 10) and *Triangulation3D2* ((2a) and (2b)). Further inspection of the respective source code locations revealed that there are two additional classes spawning threads for Delaunay triangulation. Being surprised by the fact that there is no functionality that is reused by the coordinating thread to manage the worker threads, the developer decided to consolidate that code in a single class responsible for handling the triangulation worker threads. This greatly simplifies the code and eases further maintenance of this functionality. Besides this architectural design flaw, the developer identified critical code clones in the respective methods that spawned the threads: The duplicated code contained functionality to allocate memory, spawn the thread and clean up after thread termination by freeing numerous memory locations.

The visualization technique helped to understand how triangulation threads are coordinated in BRec. Not only contribute multiple threads to the rendering functionality, but key parts of this functionality were repeatedly implemented, which further complicated an understanding of the rendering functionality.

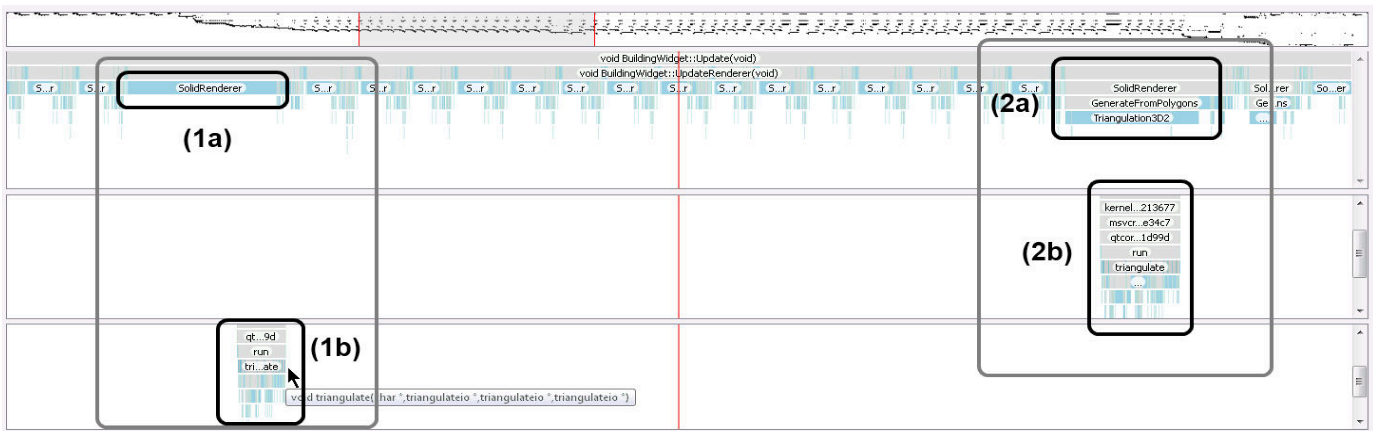


Fig. 10. Sequence visualization of BRec's rendering process: different class constructors (1a, 2a) spawn two worker threads (1b and 2b).

VII. Conclusions

Understanding and maintaining multithreaded software systems is typically much more difficult than single-threaded systems. The application of multithreading impacts development and maintenance processes in several ways: (1) System design is being complicated, (2) runtime behavior is often unpredictable and much more complex, and (3) program comprehension is aggravated, e.g., due to a weak correlation between the source code's control structures and the system's runtime behavior.

We propose a visualization technique for analyzing the runtime behavior of complex, multithreaded software systems that uses out-of-core concepts to handle large execution traces. Its usefulness is investigated by case studies on two complex software systems: Google's web browser Chromium and an industry tool for reconstructing 3D building models from laser scan data.

As future work, we plan to evaluate the impact of our visualization technique on the performance of developers. In addition, we focus on trace comparison techniques such as pattern matching or dynamic time warping. We also plan to exploit data mining techniques to identify outliers in trace data, which seems to be a promising approach to provide automated support for locating root causes of race conditions.

Acknowledgments

We want to thank Software Diagnostics GmbH (http://www.softwarediagnostics.com) for providing us with their tracing and visualization framework. Furthermore, we want to thank virtualcitySYSTEMS GmbH for providing us with their BRec system.

References

[1] Victor R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[2] Michael Bedy, Steve Carr, Xianlong Huang, and Ching-Kuang Shene. A visualization system for multithreaded programming. *SIGCSE Bulletin*, 32(1):1–5, 2000.

[3] Johannes Bohnet, Martin Koeleman, and Jürgen Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 57–64, 2009.

[4] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[5] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[6] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *IEEE International Conference on Program Comprehension*, pages 181–190, 2006.

[7] Michael Heath. Paragraph: A tool for visualizing performance of parallel programs. Technical report, University of Illinois, 1993.

[8] Jacques Chassin De Kergommeaux, Benhur De Oliveira Stein, and Montbonnot Saint Martin. Pajé: An extensible environment for visualizing multi-threaded program executions. In *European Conference on Parallel Computing*, pages 133–144, 2000.

[9] Adrian Kuhn and Orla Greevy. Exploiting the analogy between traces and signal processing. In *IEEE International Conference on Software Maintenance*, pages 320–329, 2006.

[10] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *International Conference on Software Engineering*, pages 492–501, 2006.

[11] A. D. Malony and D. A. Reed. *Instrumentation for future parallel computing systems*, chapter Visualizing parallel computer system performance, pages 59–90. ACM, 1989.

[12] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[13] G.J. Nutt, A.J. Griff, J.E. Mankovich, and J.D. McWhirter. Extensible parallel program performance visualization. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:205–211, 1995.

[14] David Lorge Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, 1994.

[15] Steven P. Reiss. Visual representations of executing programs. *Journal of Visual Languages and Computing*, 18(2):126–148, 2007.

[16] John T. Stasko. The parade environment for visualizing parallel program executions: A progress report. Technical report, Georgia Institute of Technology, 1995.

[17] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2nd edition, 2004.

[18] Richard G. Waters and Elliot Chikofsky. Reverse engineering: progress along many dimensions. *Communications of the ACM*, 37(5):22–25, 1994.

[19] Kyle Wheeler and Douglas Thain. Visualizing massively multithreaded applications with threadscope. *Concurrency and Computation: Practice and Experience*, 22:45–67, 2009.

[20] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*, pages 200–205, 1992.

```
@inproceedings{TBVD10,
author = { Jonas Trümper and Johannes Bohnet and Stefan Voigt and
Jürgen Döllner },
title = { Visualization of Multithreaded Behavior to Facilitate
Maintenance of Complex Software Systems },
booktitle = { Proceedings of the International Conference on the
Quality of Information and Communications Technology },
pages = { 325-330 },
year = { 2010 },
publisher = { IEEE Computer Society }
}
```